A Design Pattern for Recoverable Job Management

Richard Cardone
Steve Black
Christian Garcia
Anagha Jamthe
Mike Packard
Smruti Padhy
Joe Stubbs
Texas Advanced Computing Center, Austin, Texas USA, rcardone, scblack, cgarcia, ajamthe, mpackard, spadhy
jstubbs@taccu.texas.edu

ABSTRACT

Processing scientific workloads involves staging inputs, executing and monitoring jobs, archiving outputs, and doing all of this in a secure, repeatable way. Specialized middleware has been developed to automate this process in HPC, HTC, cloud, Kubernetes and other environments. This paper describes the Job Management (JM) design pattern used to enhance workload reliability, scalability and recovery. We discuss two implementations of JM in the Tapis Jobs service, both currently in production. We also discuss the reliability and performance of the system under load, such as when 10,000 jobs are submitted at once.

CCS CONCEPTS

• **Software reliability**; • **Middleware**; • **Design patterns**;

KEYWORDS

HPC, Cloud computing, Job management

ACM Reference Format:
Richard Cardone, Steve Black, Christian Garcia, Anagha Jamthe, Mike Packard, Smruti Padhy, and Joe Stubbs. 2022. A Design Pattern for Recoverable Job Management. In *Practice and Experience in Advanced Research Computing (PEARC '22), July 10–14, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3491418.3535136

1 INTRODUCTION

Processing scientific workloads involves staging inputs, executing and monitoring jobs, archiving outputs, and doing all of this in a secure, repeatable way. The likelihood of transient errors increases during large data transfers and long running jobs, so a robust workload manager needs to recover from temporary blocking conditions, such as network delays and processing spikes.

Tapis [12],[14],[15] middleware supports scientific workloads on any SSH accessible system, including HPC, HTC and cloud systems. Tapis is a set of REST-based microservices that applications use to interact with storage and execution systems. This paper focuses on the internal design of the Jobs service, which is the Tapis component responsible for managing and executing jobs, and how its design is broadly applicable to software that runs asynchronous workloads. These workloads are often multi-node, MPI jobs that run under batch schedulers such as Slurm.

The contributions of this paper are:

- The introduction of a generally applicable Job Management (JM) design pattern [5], its main data structures and its operational semantics.
- Two production implementations (Jobs v2 and v3) of the JM design pattern.
- Performance results that indicate the effectiveness of the design.

2 JOB MANAGEMENT (JM) DESIGN PATTERN

2.1 Job Workers

Figure 1 depicts the main components of the Job Management (JM) design pattern. On the left, a Job Worker receives incoming job requests on its work queue from the Jobs front end (not shown). A state machine drives job processing, starting in the ACCEPT state and terminating in FINISHED, FAILED or CANCELED. Each state transition triggers an action, which either completes successfully, fails in an unrecoverable way, or fails in a recoverable way. The Job Table tracks the state of all jobs, each identified by a unique *jobId*.

The Work Queue and Job Table are typically backed by persistent storage. Different implementations can support different levels of durability and reliability. For example, message delivery can be done on a best effort, at least once, or exactly once basis. Similarly, implementations choose the level of concurrency they support. The number of worker processes, threads per process, work queues and workers per queue are implementation dependent.

2.2 Recovery Processing

Figure 1 depicts the Job Recovery Process on the right. When a Job Worker experiences a recoverable error, it writes a recovery message to the Recovery Queue and sets the job's *state* to BLOCKED. Recoverable errors are transient conditions that will eventually clear, though determining if an error is recoverable is implementation dependent. For example, quota violations, short network outages and communication timeouts are often recoverable.

PEARC '22, July 10–14, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9161-0/22/07…$15.00
https://doi.org/10.1145/3491418.3535136

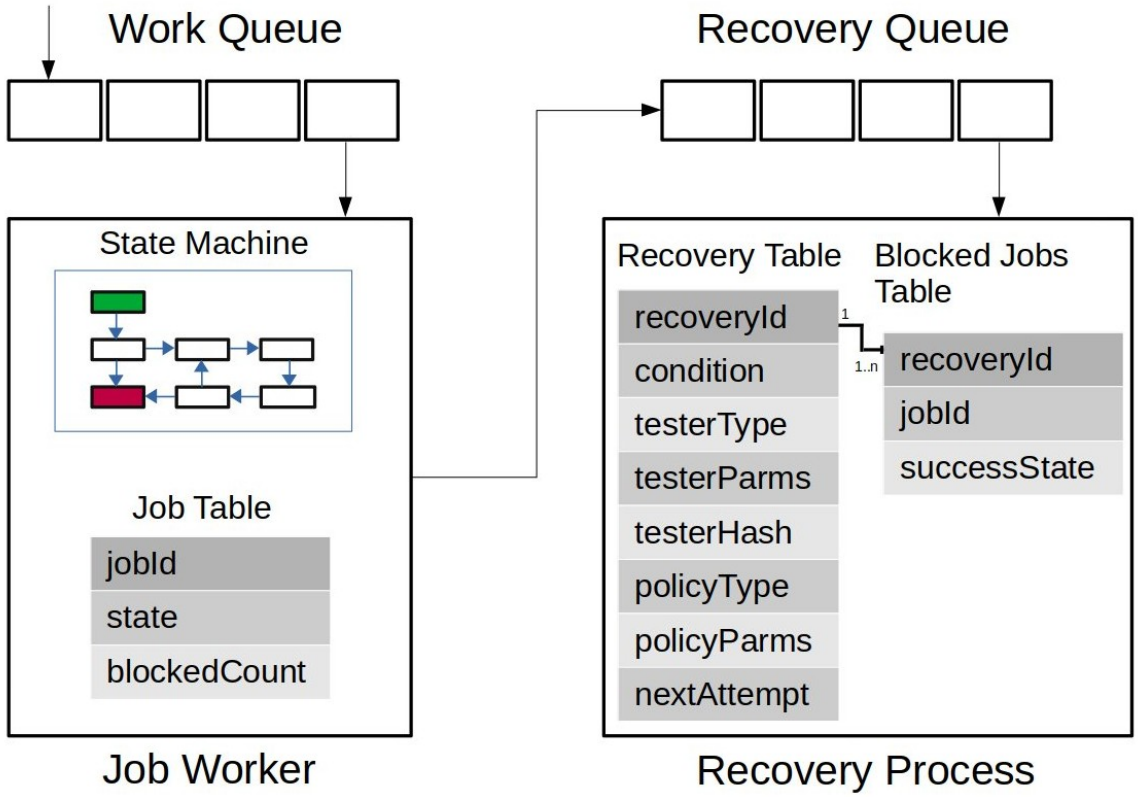

Figure 1: Job Management Design Pattern

The Recovery Process reads its Recovery Queue and writes each blocked job to its tables. The number of queues and recovery processes, as well as the durability of queues and tables, are implementation dependent. There is a one-to-many relationship between the Recovery Table and the Blocked Job Table: each recovery record is linked to one or more blocked job records. All jobs blocked on the same condition are linked to the same recovery record.

2.3 Exceptions, Policies, Testers and Throttles

Job processing can be interrupted for many reasons, such as a system going offline, a DNS timeout, a user suspending the job, or the application becoming disabled. When such conditions are encountered, workers perform an analysis to determine if the error is recoverable. If so, the worker constructs a *recoverable exception* that includes contextual information. The exception is thrown and routines higher on the call stack can add more contextual information.

Eventually, the exception bubbles up to the worker's recovery handler, which creates a recovery message. Associated with each exception type is a *policy* and a *tester*. Testers implement the code that detects when a blocking condition clears; policies determine when and for how long tests run. The exception's contextual information is added to the message's *testerParms* and *policyParms* fields and the *condition* field reflects the exception type. As noted, the worker then sets the job's state to BLOCKED and posts the message to the Recovery Queue.

The Recovery Process searches its Recovery Table for a record with the same *testerHash* as the message it receives. The *testerHash* is a hash of the *condition*, *testerType* and *testerParms* that groups together jobs waiting for the same condition to clear. If the *testerHash* is not found, a new Recovery Table record is inserted, a new Blocked Table record is linked to it, and the *nextAttempt* time is calculated using *policyType* and *policyParms*. If the *testerHash* is found, a new Job Blocked Table record is linked to the existing recovery record.

Though not required by the design pattern, Jobs uses a multithreaded Recovery Process that has a *main thread* to read the Recovery Queue and a *test thread* to detect when tests should run. The main thread reads messages from the queue and updates the tables. The test thread monitors *nextAttempt* times and runs tests as scheduled.

When a test detects a cleared blocking condition, it removes one or more jobs from the Blocked Jobs Table, resets their states to *successState*, and requeues them on the Work Queue. The *successState* is the restart state that the worker designated when it blocked. To avoid thrashing, a throttle limits the number of jobs requeued at one time. Recovery Table entries with no blocked jobs are deleted.

3 TAPIS V2 IMPLEMENTATION OF JM

The Tapis v2 [13] Jobs service is the first JM implementation. It went into production in June 2019 as a drop-in replacement for the prior Agave [4] Jobs service. As a multi-tenant application, Jobs v2 assigns each tenant its own Work Queue and dedicated set of workers. Workers are separate Java programs configured with 30-100 threads, each thread processing a job independently. Jobs can scale up by adding worker threads and/or adding new workers. A separate Java recovery program is also run for each tenant, each with its own Recovery Queue. We'll see that Jobs v3 takes a different approach to multi-tenancy.

All queues and tables are backed by persistent storage in RabbitMQ and MySQL, respectively. Jobs in the Work Queue are not removed until the job reaches a terminal state. Upon job arrival, workers check the database to distinguish between new and restarted jobs.

There are cases where atomicity spanning RabbitMQ and MySQL operations is required for correctness, such as when a worker sets a job's state to BLOCKED and then puts the job on the recovery queue. Jobs coordinates RabbitMQ and MySQL by using a combination of consistency checking, rollbacks and repair actions. In practice, this approach works well on our relatively stable computing infrastructure and avoids the complexity of federated transactions.

Jobs v2 implements two recovery policies, constant backoff and stepwise backoff. The former specifies testing at fixed intervals for a fixed amount of time; the latter specifies testing at fixed intervals and durations for each step, but any number of steps can be configured. Constant backoff is appropriate for lightweight testing, such as polling another Tapis service. Stepwise backoff is appropriate when testing should decrease in frequency over time, such as when retrying SSH connections that previously timed out. Stepwise backoff is similar to exponential backoff policies, but it provides simple, direct control of the backoff algorithm.

Currently, Job tester routines attempt recovery from connection, authentication and transmission errors, as well as internal conditions such as "soft" quota violations, disabled systems and disabled applications. Soft quotas delay jobs but do not abort them. For example, when a Tapis system limits the number of jobs a user can concurrently run, excess jobs are held until the quota is no longer exceeded. Schedulers such as Slurm usually reject jobs in such situations. Tapis does, however, treat some quotas violations as unrecoverable, such as disk quota full.

4 TAPIS V3 IMPLEMENTATION OF JM

The Tapis v3 Jobs service went into production February 2022 and represents a complete rewrite and a new implementation of the JM design pattern. Jobs v3 addresses new requirements, such as containerized apps and multi-site support, and is not backward compatible with v2.

Jobs v3 uses one Work Queue, Recovery Queue, Job Worker and Recovery Process for all tenants at a site, making it more cloud friendly than v2. PostgreSQL replaces MySQL. Recovery exceptions, policies and testers largely carry over from v2. In both v2 and v3, the number of unblocked jobs simultaneously resubmitted to the Work Queue is limited to 10. This simple throttling avoids jobs quickly reblocking due to the original threshold being crossed again.

New to v3 is a throttle that establishes a 2 second sliding window in which at most 10 jobs can begin processing. If the limit is exceeded, a job's processing will be randomly delayed for between 1 and 3 seconds. Under high load, this short delay provides enough time for concurrent data structures to stabilize and for database updates to propagate.

Also new to v3 is a throttle that establishes a 2 second sliding window in which at most 8 jobs can be launched on the same host. When the threshold is exceeded, the launch is delayed for a random number of seconds between 3 and 63. This reduces the likelihood that an execution system will be temporarily overloaded by many network connections and program invocations.

5 PERFORMANCE

After nearly three years in production, Jobs v2 processed over 155,000 jobs. Of the 44,905 jobs that went into recovery at least once, 30,964 completed successfully; 5,209 were canceled by a user; 8,697 failed; and 35 were still processing when the snapshot was taken. It's clear that recovery processing is an ordinary and essential part of job execution.

In Tapis v3, we submitted 10,000 jobs at once to observe the new system under load.[1] The jobs ran for about 10 hours on a VM, though all they did was sleep for a random number of seconds between 15 and 90 (average ~52) and then terminate. One job failed, the rest succeeded. The execution system ran up to 50 jobs at once. The theoretical optimal scheduling would run 200 batches of 50 jobs, one after the other, all jobs starting and ending at the same time in each batch. Assuming no overhead and perfect scheduling, the ideal total runtime would be about 173 minutes. Actual runtime was about 600 minutes, or .29 of the ideal.

The difference between ideal and actual runtimes is due to non-uniform job durations; computational, database, queuing and network overhead; throttling when under heavy load to avoid overwhelming the execution system; monitoring that detects job completion after the fact via polling; and the unblocking of jobs in batches of no more than 10 to avoid queue thrashing. The main takeaway, however, is that JM delivers steady, controlled progress even under heavy load.

6 RELATED WORK

JM is a design pattern for scalable applications where reliability is a first class concern. It incorporates work queues, message passing and a tunable, recovery mechanism to achieve high concurrency and reliable throughput in the presence of transient errors. Other scalability design patterns, such as actors [6], reactors [11] and schedulers, provide concurrency but no notion of recoverable errors. Actors incorporate message passing, reactors run callback routines from an event queue, and schedulers control executions using time-slicing and other techniques, but recovery using policies and testers are not standard components of these patterns. It is feasible, however, to add a JM style recovery mechanism to these other concurrency designs, especially to actors and reactors which already use message passing.

[1] Jobs v2 also passed the 10,000 job test, though details of those runs are no longer available.

Like Tapis, Apache Airavata [1], HubZero [7],[8] and Open OnDemand [9] manage scientific computations. These facilities often support workflows, projects, experiments, application authoring, collaboration environments, and GUIs. Tapis provides APIs upon which such abstractions can be built, but does not itself implement them. DesignSafe [3] and Cyverse [2], for example, are Web-based collaborative environments built on top of Tapis's computational capabilities. Another distinguishing trait of Tapis is its ability to run jobs on any host accessible via SSH—including user laptops—with no installation required.

7 CONCLUSION

The JM design pattern provides a flexible, reliable way to manage asynchronous jobs and recover from transient errors that would otherwise cause failures. The pattern is applicable to any long running service that can restart computations interrupted by temporary blocking conditions. The two Tapis implementations of JM provide years of experience with real workloads and load tests that validate the design's resiliency when stressed.

In the future, we'd like to provide stronger atomicity guarantees when coordinating queue and database updates. For instance, the PostgreSQL pg_amqp plugin [10] allows database triggers to issue RabbitMQ commands inside transactions. Alternatively, Jobs could commit a database transaction only if the subsequent queue operation succeeds. Another future work item is to collect more test data using different throttle settings to further optimize and bulletproof Jobs.

ACKNOWLEDGMENTS

This work is supported by National Science Foundation grants 1931439 and 1931575.

REFERENCES

[1] Apache Airavata (2022). https://airavata.apache.org/index.html. Accessed 25 Mar 2022.
[2] Cyverse (2022). https://cyverse.org/. Accessed 25 Mar 2022.
[3] DesignSafe (2022). https://www.designsafe-ci.org/. Accessed 25 Mar 2022.
[4] Dooley, R., *et al.*: Software-as-a-Service: the iPlant foundation API. In: 5th IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS). IEEE (2012)
[5] Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software (1995). Reading, Mass: Addison-Wesley.
[6] Hewitt, Carl; Bishop, Peter; Steiger, Richard (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.
[7] HubZero (2022). https://hubzero.org/. Accessed 25 Mar 2022.
[8] McLennan, M., and Kennell, R. (2010). "HUBzero: a platform for dissemination and collaboration in computational science and engineering." Computing in Science & Engineering 12.2 (2010): 48-53.
[9] Open OnDemand. https://openondemand.org/. Accessed 25 Mar 2022.
[10] Roy, G.: RabbitMQ in Depth (2017). Chapter 10. Shelter Island, NY: Manning.
[11] Schmidt, Douglas *et al.* Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Volume 2. Wiley, 2000.
[12] Stubbs J. *et al.* (2021) Tapis: An API Platform for Reproducible, Distributed Computational Research. In: Arai K. (eds) Advances in Information and Communication. FICC 2021. Advances in Intelligent Systems and Computing, vol 1363. Springer, Cham. https://doi.org/10.1007/978-3-030-73100-7_61
[13] Tapis v2 documentation. https://tacc-cloud.readthedocs.io/projects/agave. Accessed 25 Mar 2022.
[14] Tapiis v3 documentation. https://tapis.readthedocs.io. Accessed 25 Mar 2022.
[15] Tapis v3 APIs. https://tapis-project.github.io/live-docs. Accessed 25 Mar 2022.